Surviving Sensor Network Software Faults

Yang Chen (University of Utah)
Omprakash Gnawali (USC, Stanford)
Maria Kazandjieva (Stanford)
Philip Levis (Stanford)
John Regehr (University of Utah)

22nd SOSP
October 13, 2009

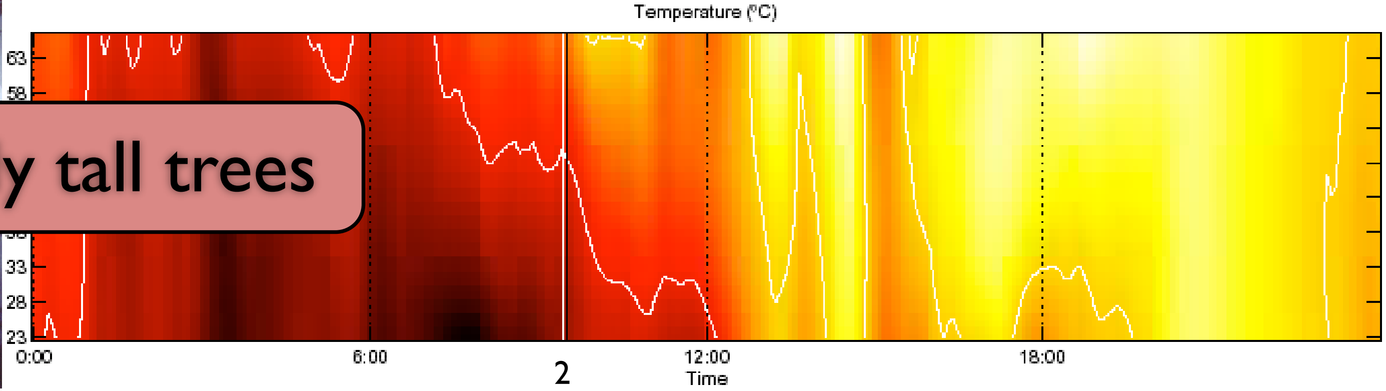
In Atypical Places for Networked Systems

Forest fires

Volcanoes

Landmarks

Really tall trees

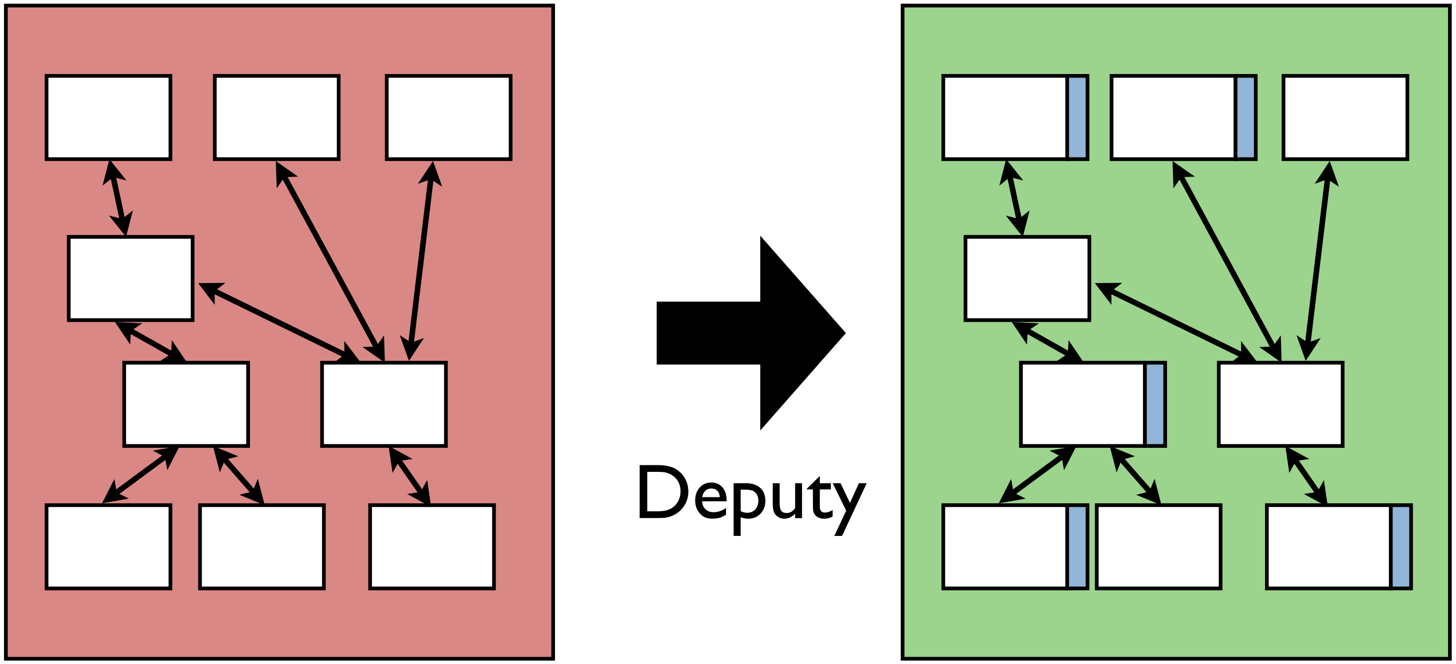


Challenges

- Operate unattended for months, years
- Diagnosing failures is hard
    - Input is unknown, no debugger
- Memory bugs are excruciating to find
    - No hardware memory protection

Safe TinyOS

(memory safety)



Safety Violation

- Lab: blink LEDs, spit out error message
- Deployment: reboot entire node (costly!)
  - Lose valuable soft state (e.g., routing tables)
    - ‣ *takes time and energy to recover*
  - Lose application data
    - ‣ *unrecoverable*

Neutron

- Changes response to a safety violation
- Divides a program into recovery units
- Precious state can persist across a reboot
- Reduces the cost of a violation by 95-99%
- Applications unaffected by kernel violations
- Near-zero CPU overhead in execution
- Works on a 16-bit low-power microcontroller

Outline

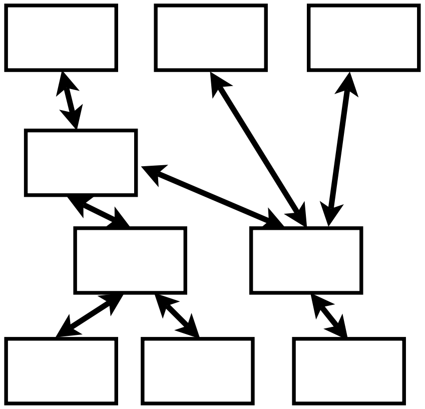
- Recovery units
- Precious state
- Results
- Conclusion

Outline

- Recovery units
- Precious state
- Results
- Conclusion

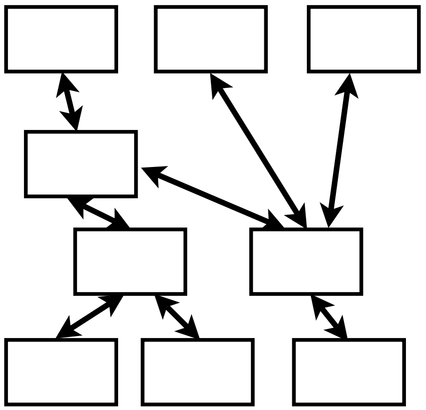
A TinyOS Program

- Graph of software components
  - Code and state, statically instantiated
  - Connections typed by interface
  - Minimal state sharing

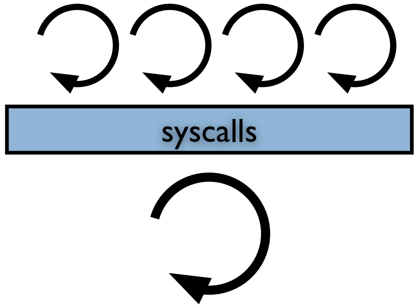


A TinyOS Program

- Graph of software components
  - Code and state, statically instantiated
  - Connections typed by interface
  - Minimal state sharing



- Preemptive multithreading
  - Kernel is non blocking, single-threaded
  - Kernel API uses message passing



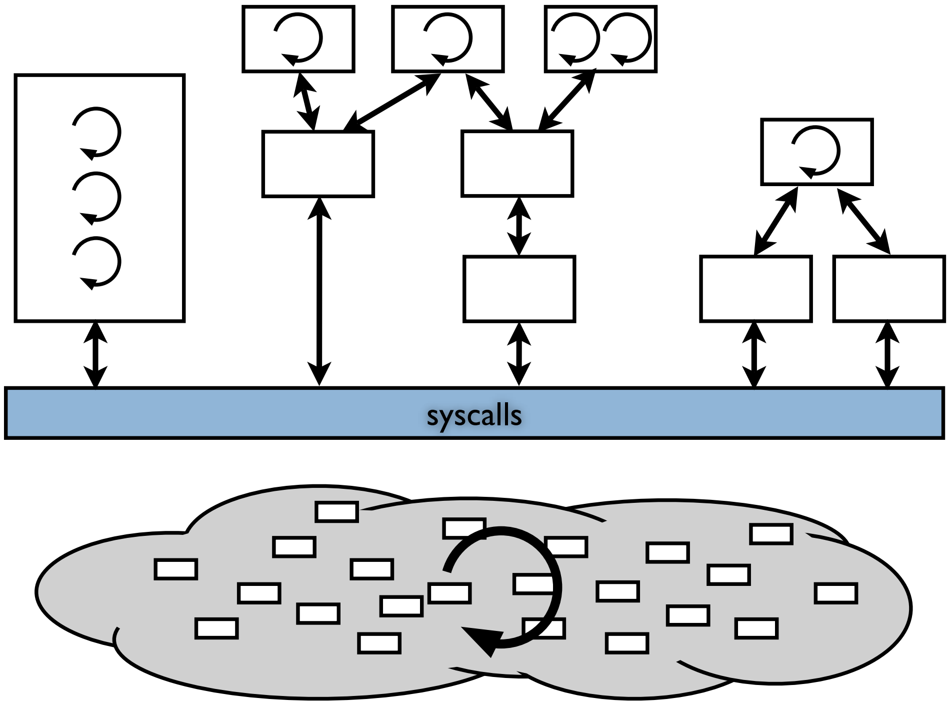
Recovery Units

- Separate program into independent units
- Infer boundaries at compile-time using:
  1. A unit cannot directly call another
  2. A unit instantiates at least one thread
  3. A component is in one unit exactly
  4. A component below syscalls is in the kernel unit
  5. The kernel unit has one thread

Recovery Units

Application Threads

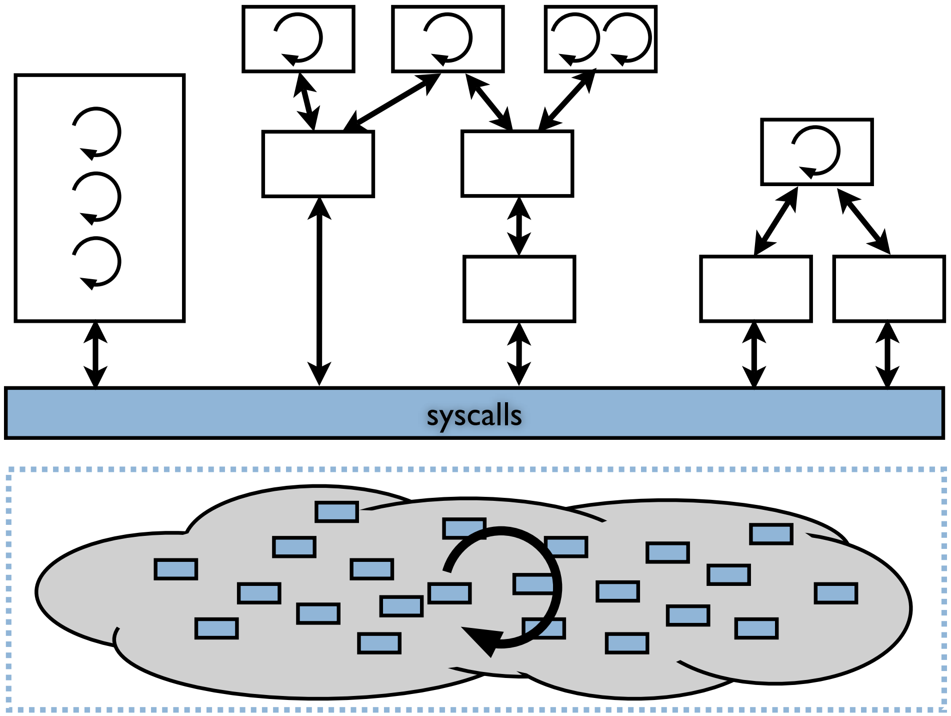
Kernel Thread



Recovery Units

Application Threads

Kernel Thread

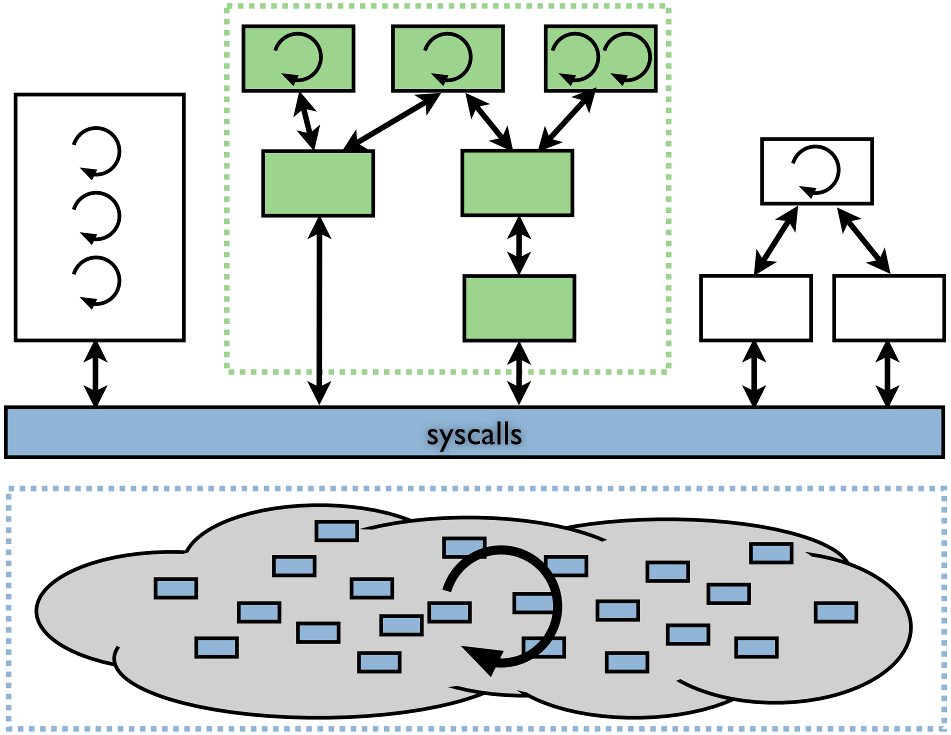


syscalls

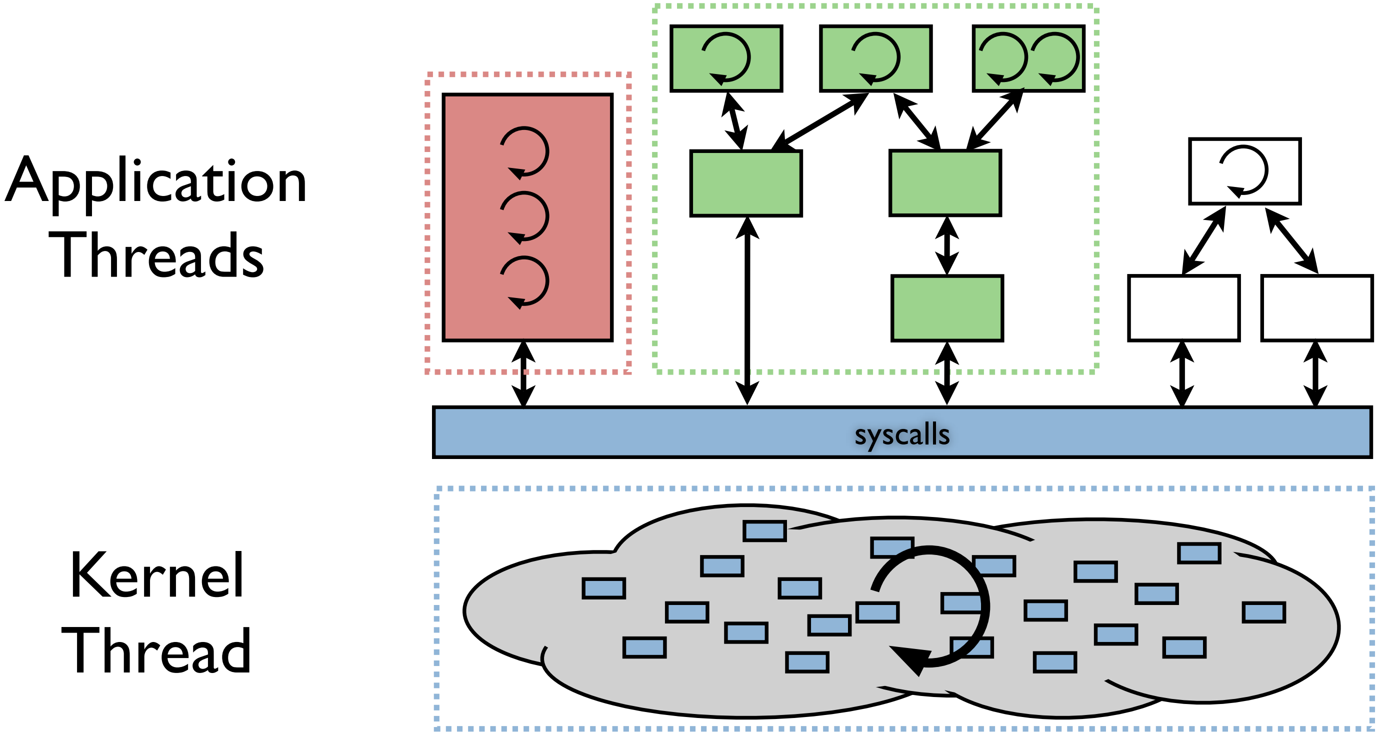
Recovery Units

Application Threads

Kernel Thread



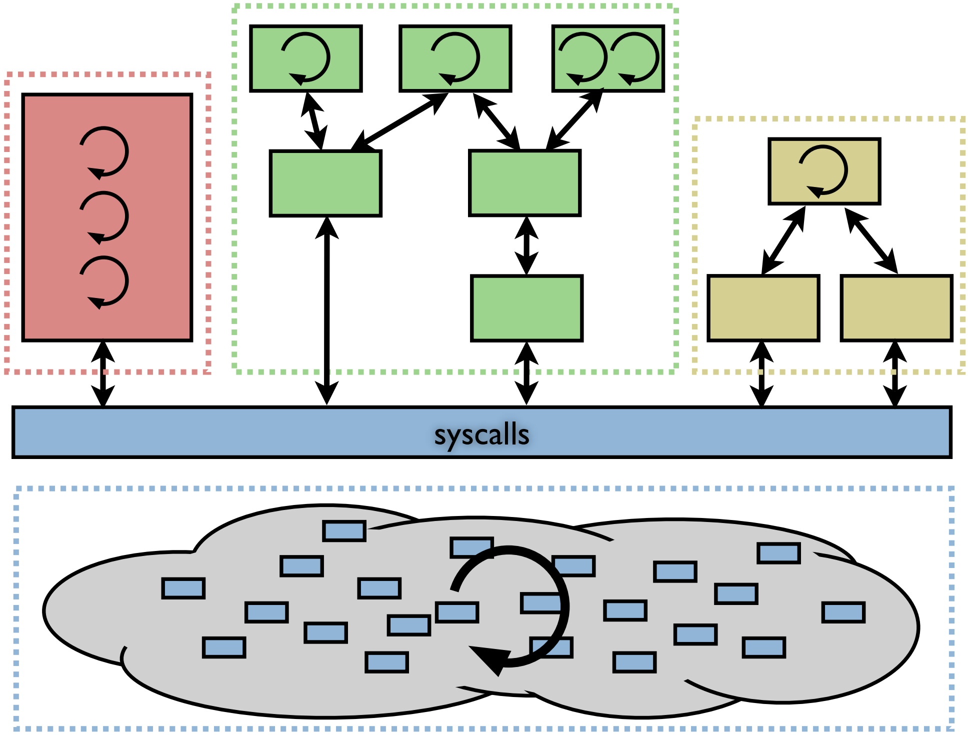
Recovery Units



Recovery Units

Application Threads

Kernel Thread

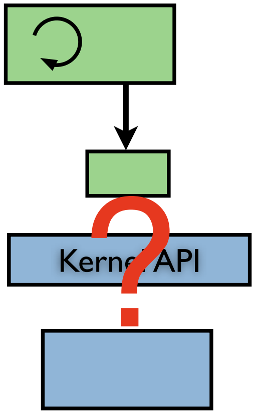


Rebooting Application Units

- Halt threads, cancel outstanding syscalls
- Reclaim malloc() memory
- Re-initialize RAM
- Restart threads

Canceling System Calls

- Problem: kernel may still be executing prior call
  - Next call will return EBUSY
- Pending flag in syscall structure
- Block if flag is set
- On completion, issue new syscall



Memory

- Allocator tags blocks with recovery unit
  - On reboot, walk the heap and free unit’s blocks
  - Must wait for syscalls that pass pointers to complete before rebooting
- On reboot, re-run unit’s C initializers
  - Each unit has its own .data and .bss
- Restart application threads

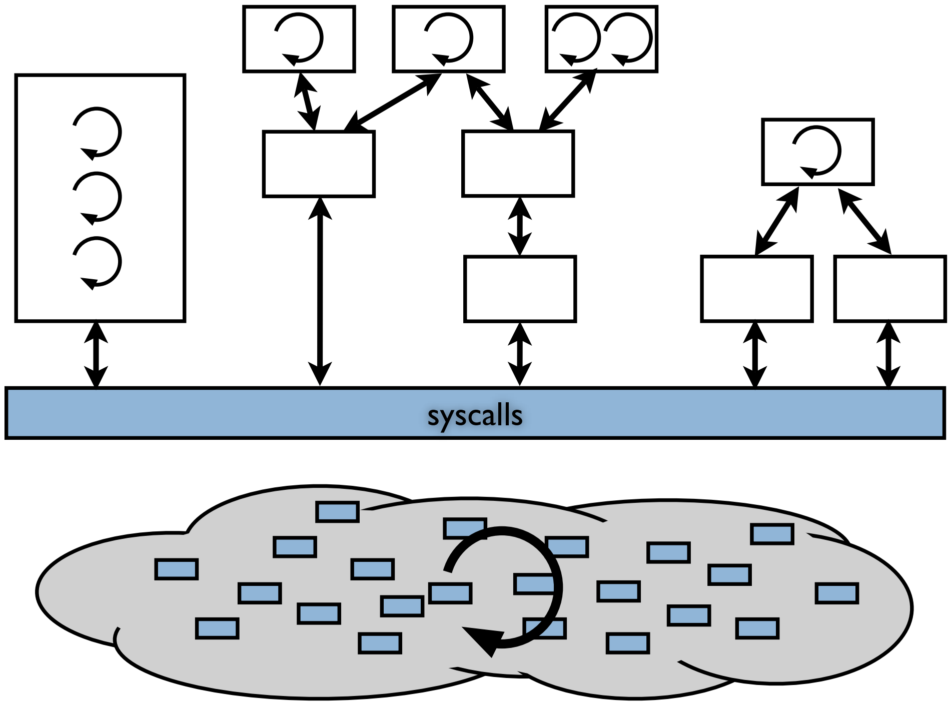
Kernel Unit Reboot

- Cancel pending system calls with ERETRY
- Reboot kernel
  - Maintain thread memory structures
- Applications continue after kernel reboots

Outline

- Recovery units
- Precious state
- Results
- Conclusion

Coupling



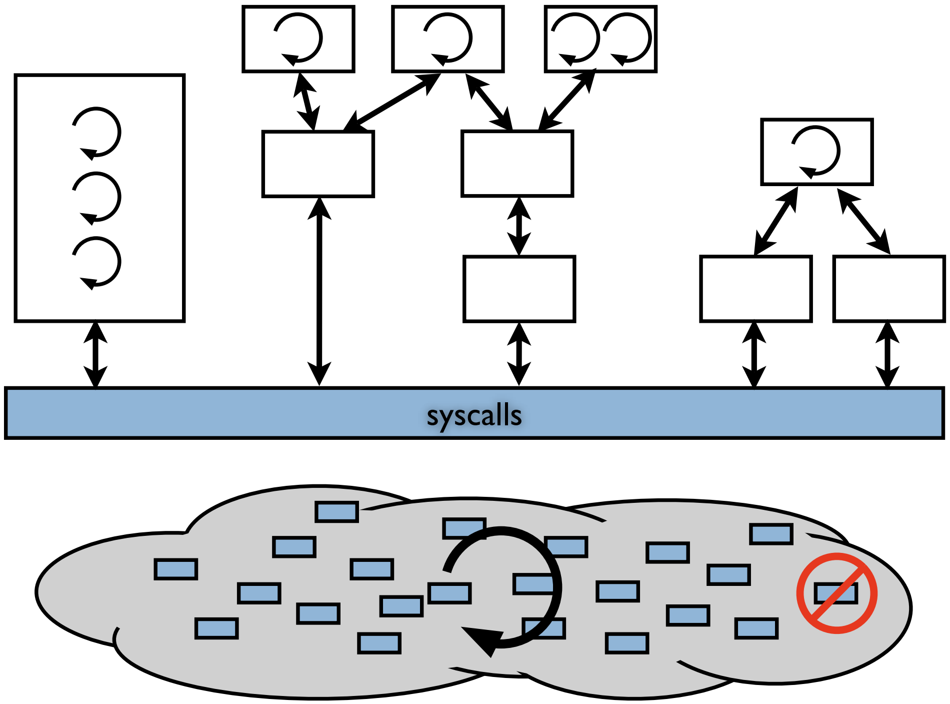
Application Threads

syscalls

Kernel Thread

Coupling

Application Threads

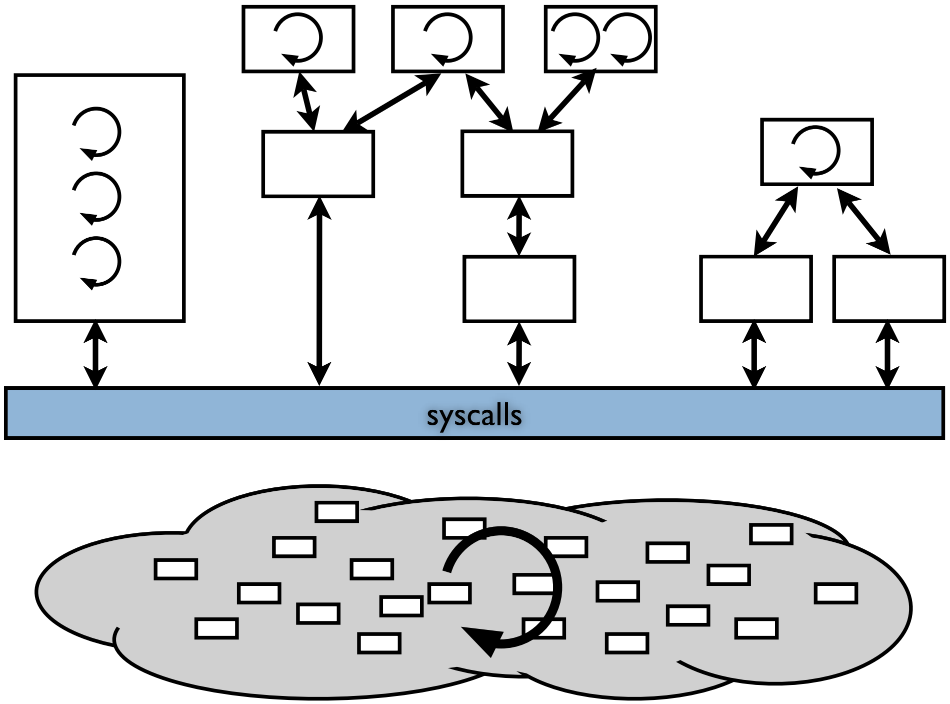


Kernel Thread

Coupling

Application Threads

Kernel Thread



Precious State

- Components can make variables "precious"
- Precious groups can persist across a reboot
  - Compiler clusters all precious variables in a component into a precious group
  - Restrict what precious pointers can point to

```
TableItem @precious table[MAX_ENTRIES];
uint8_t @precious tableEntries;
```

Persisting

- Precious variables must be accessed in `atomic{}` blocks
  - Only current thread can be cause of violation
- Static analysis determines tainted variables
  - Tainted precious state does not persist on violation

Persisting Variables

- If memory check fails, reboot unit
  - Reset current stack, re-run initializers, zero out .bss, restore variables
  - Need space to store persisting variables
- Simple option: scratch space, wastes RAM
- Neutron approach: place on stack
  - Stack has been reset
  - Often smaller than worst-case stack

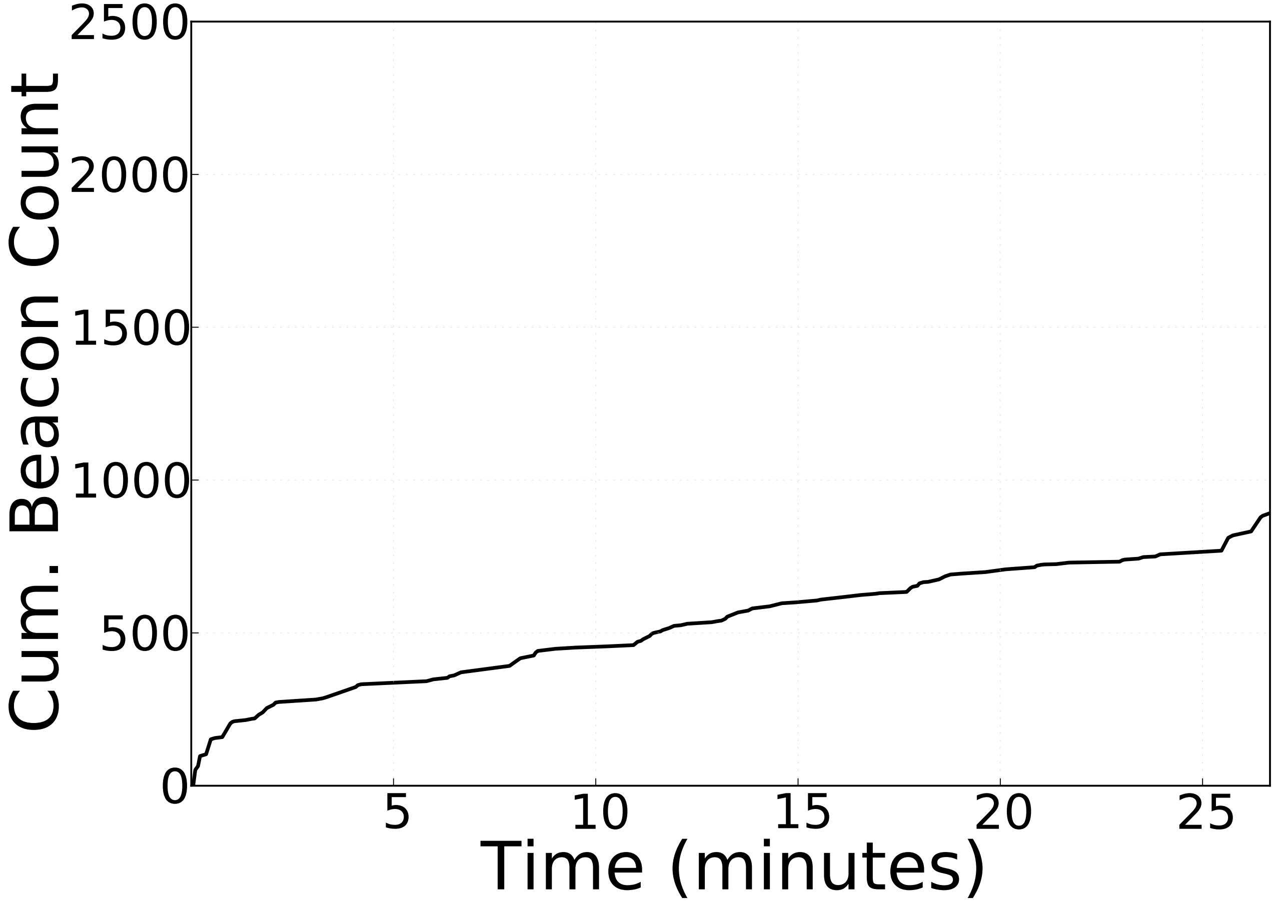
Outline

- Recovery units
- Precious state
- Results
- Conclusion

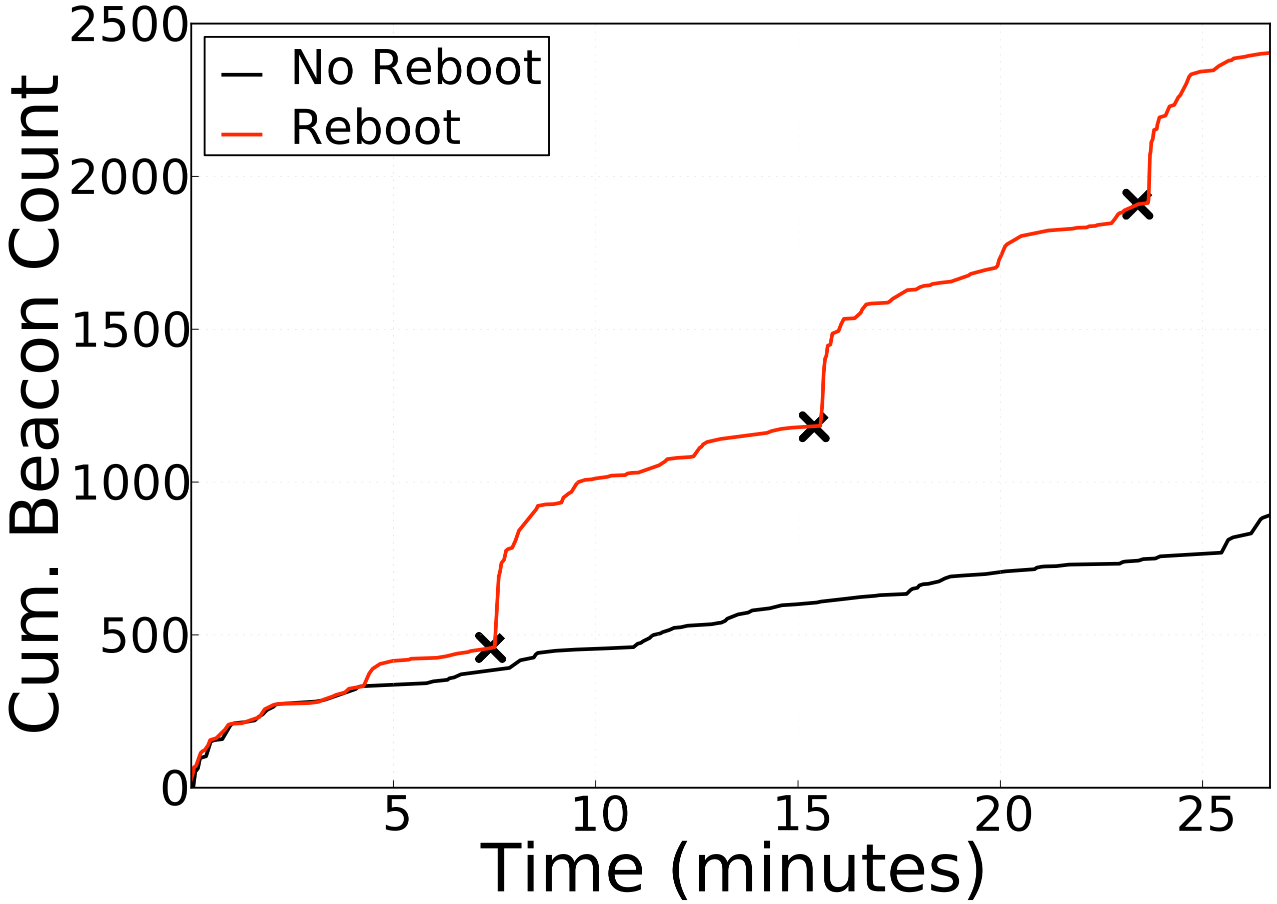
Methodology

- Evaluate cost of a kernel violation in Neutron compared to Safe TinyOS
- Three libraries, 55 node testbed (Tutornet)
    - Collection Tree Protocol (CTP), 5 variables
    - Flooding Time Synch Protocol (FTSP), 7 variables
    - Tenet bytecode interpreter in the paper
- Quantifies benefit of precious state

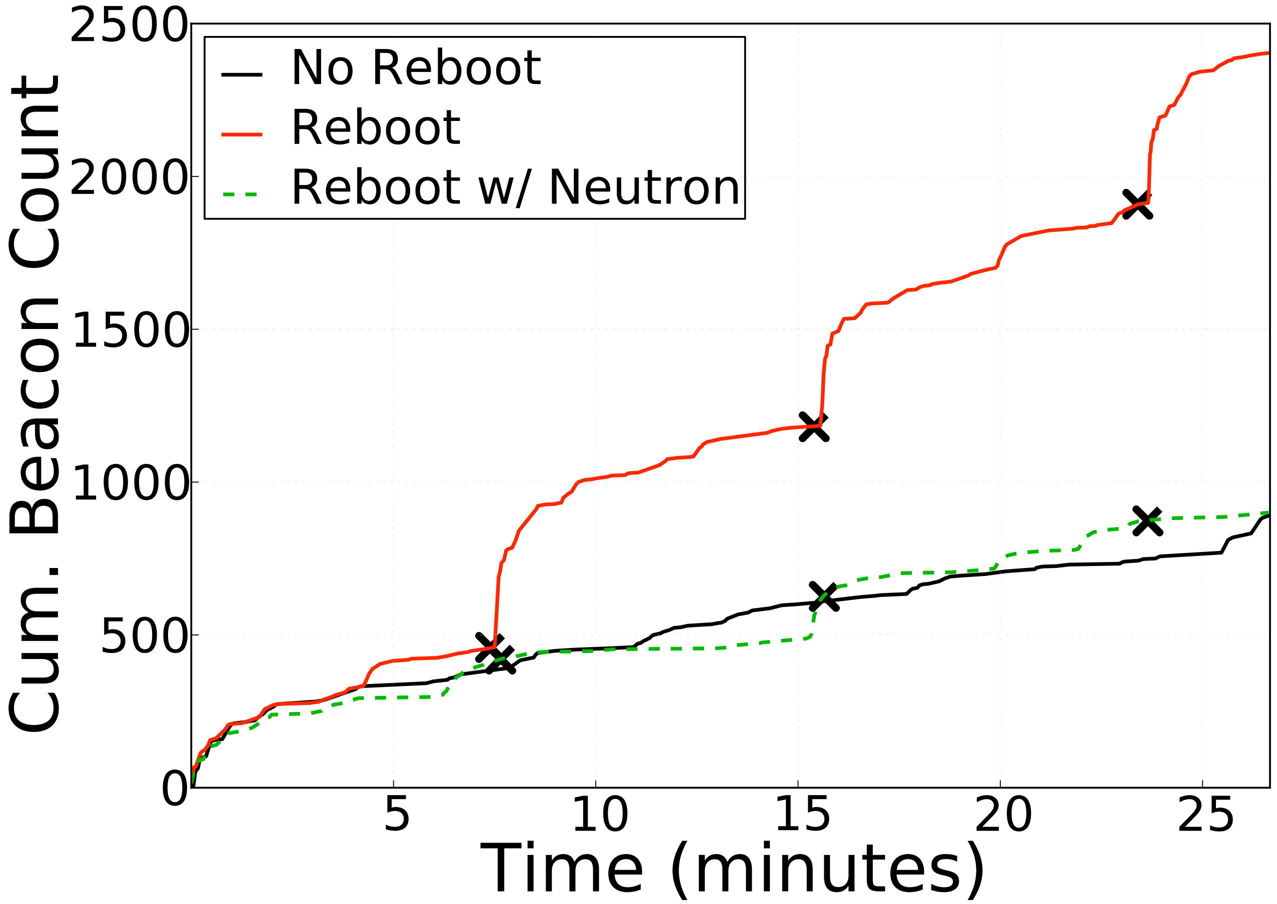
Kernel Reboot: CTP



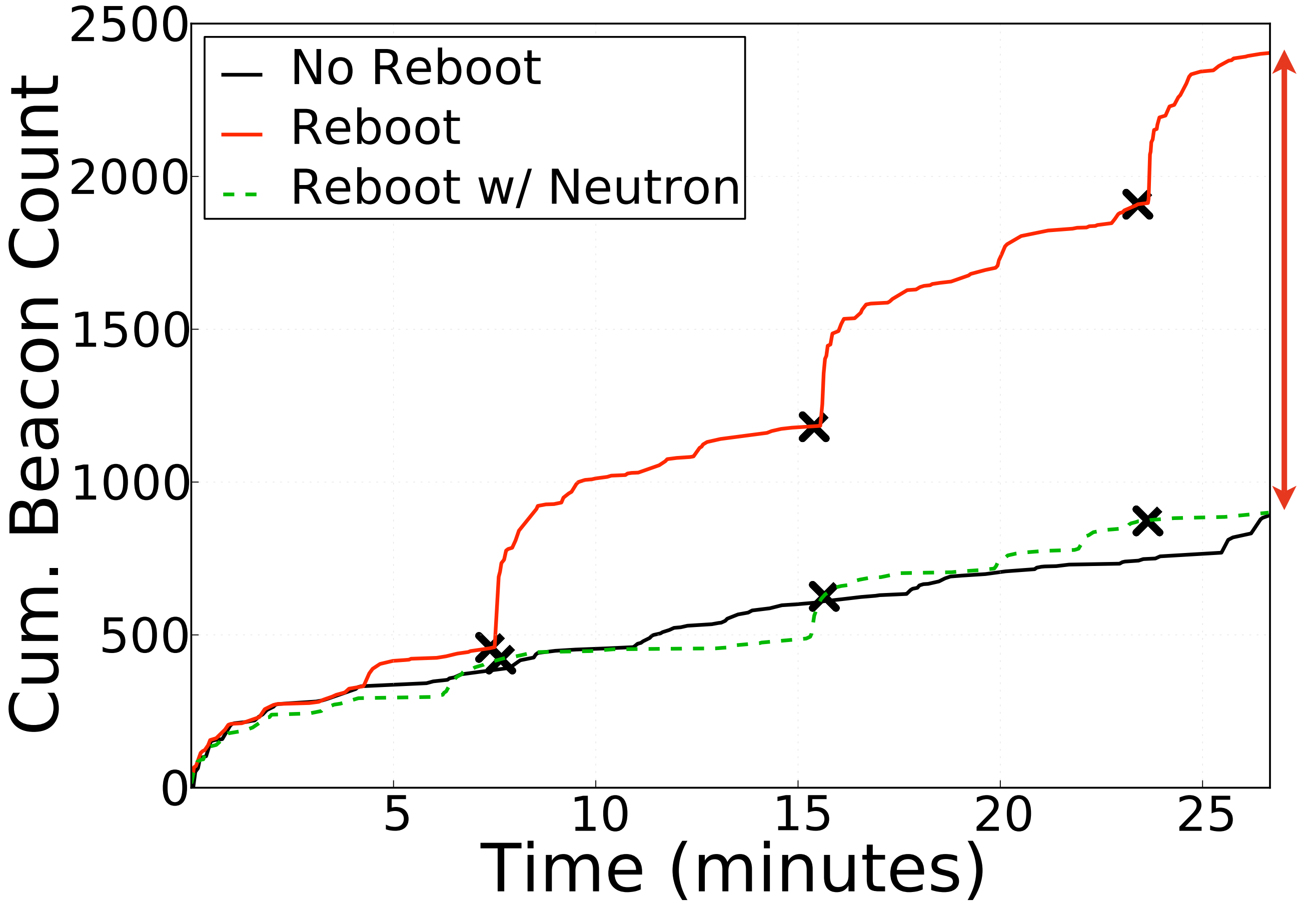
Kernel Reboot: CTP



Kernel Reboot: CTP

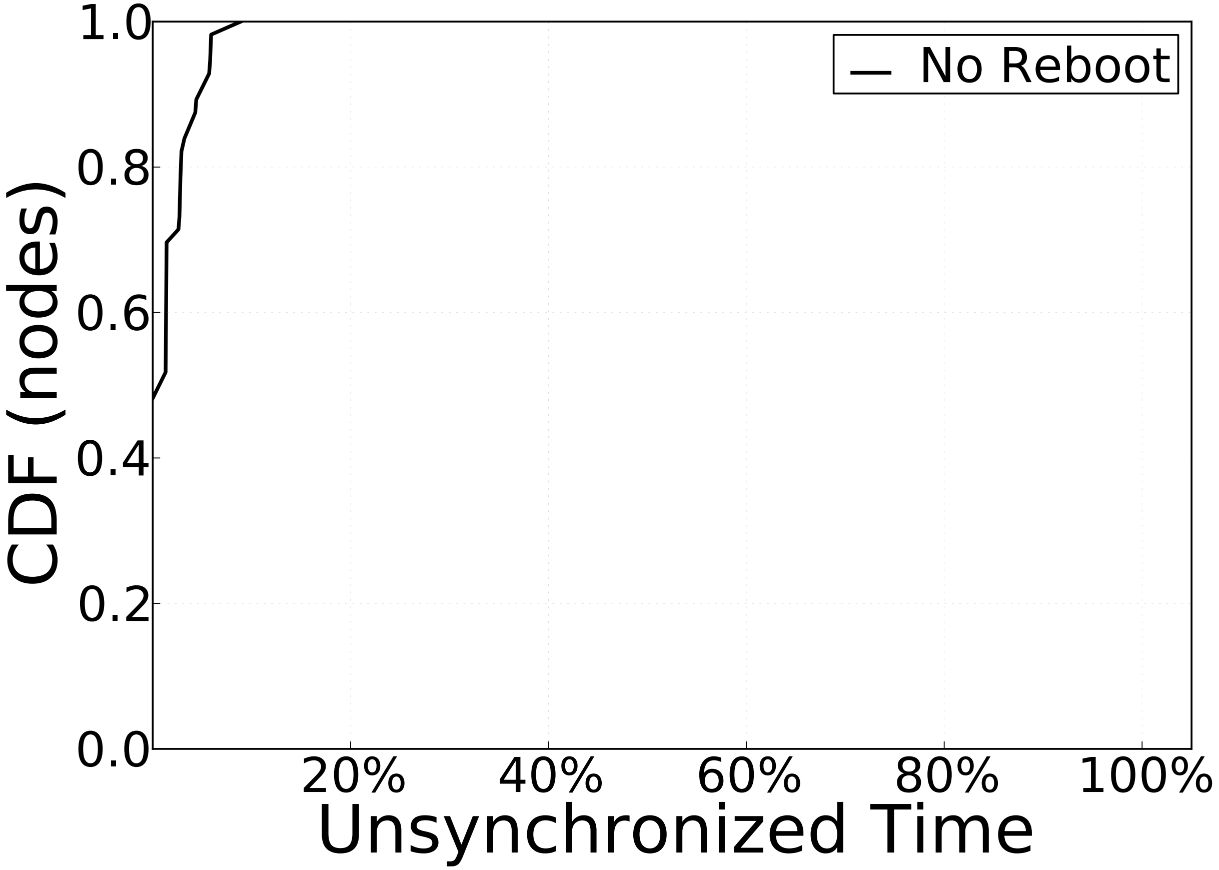


Kernel Reboot: CTP

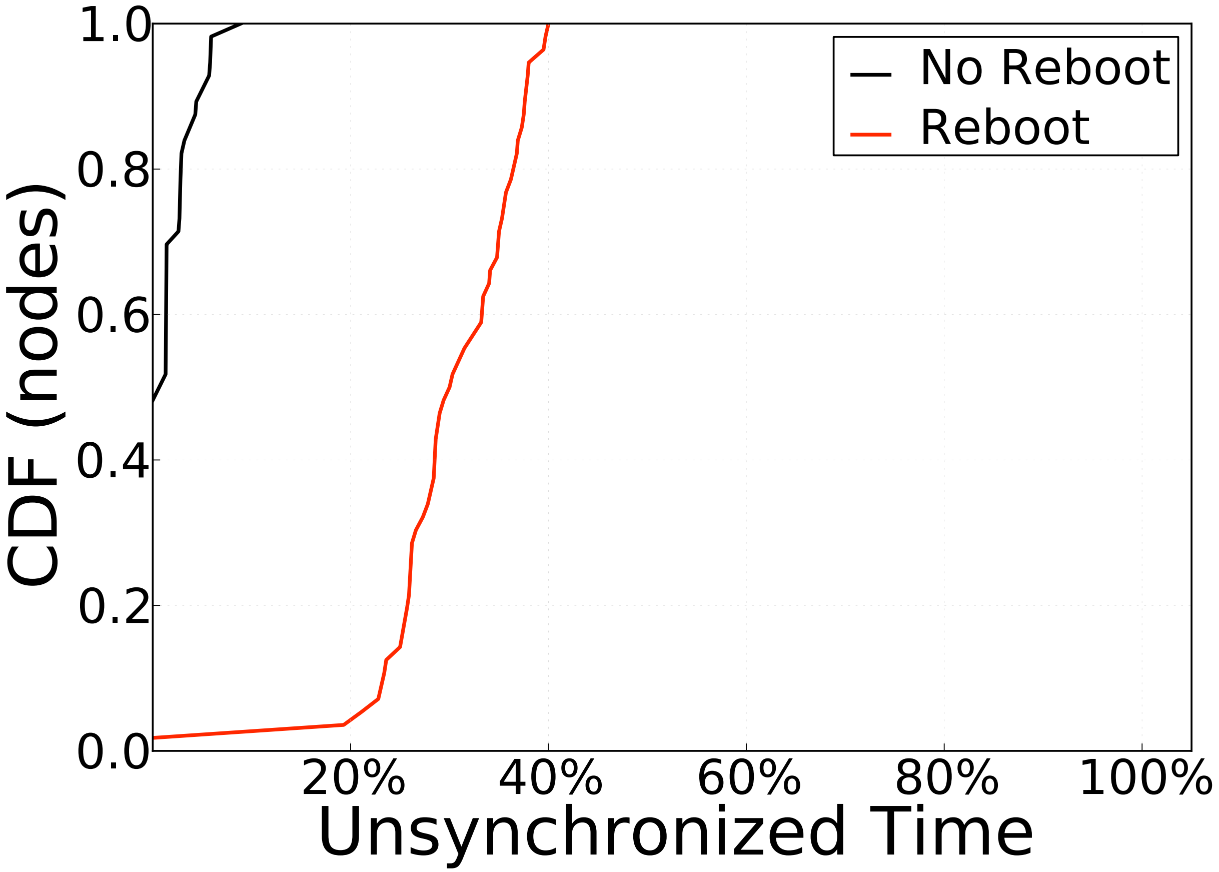


99.5% reduction

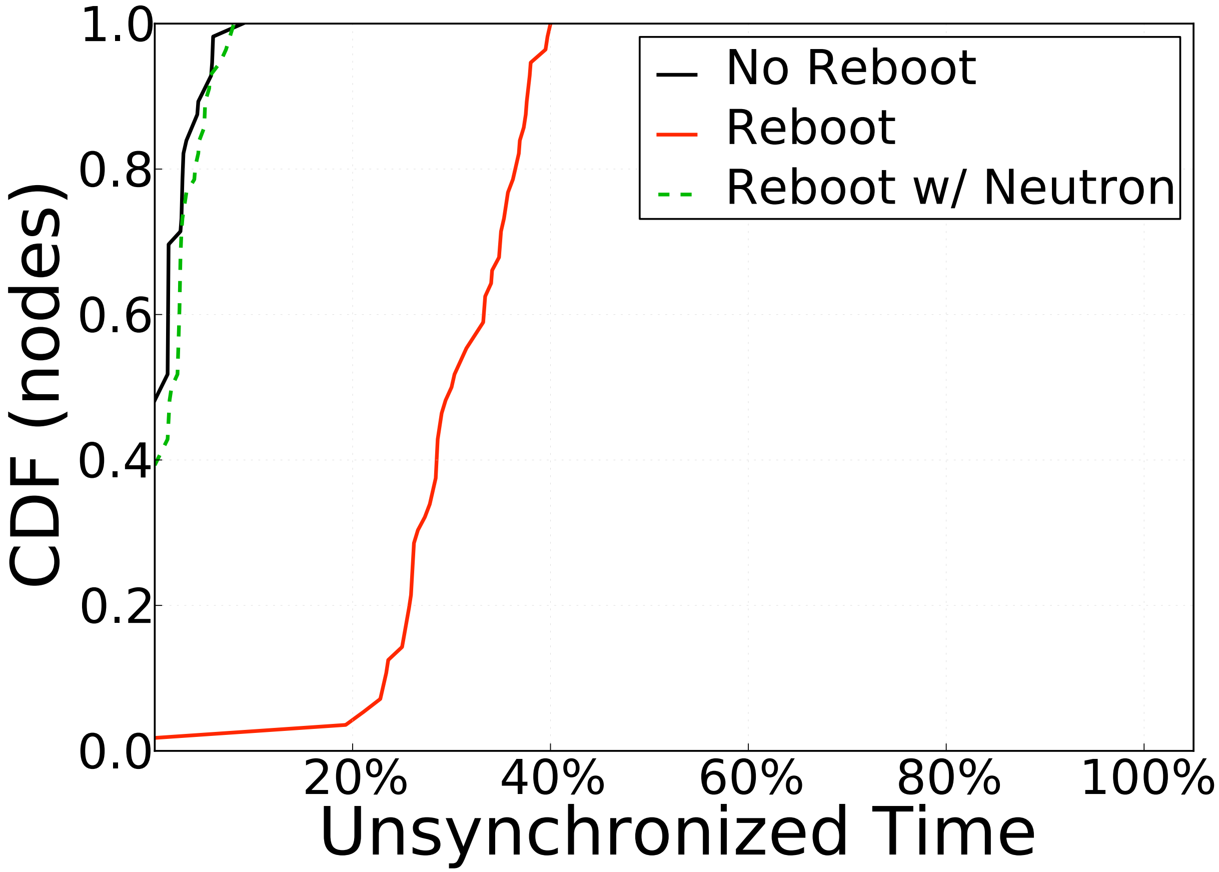
Kernel Reboot: FTSP



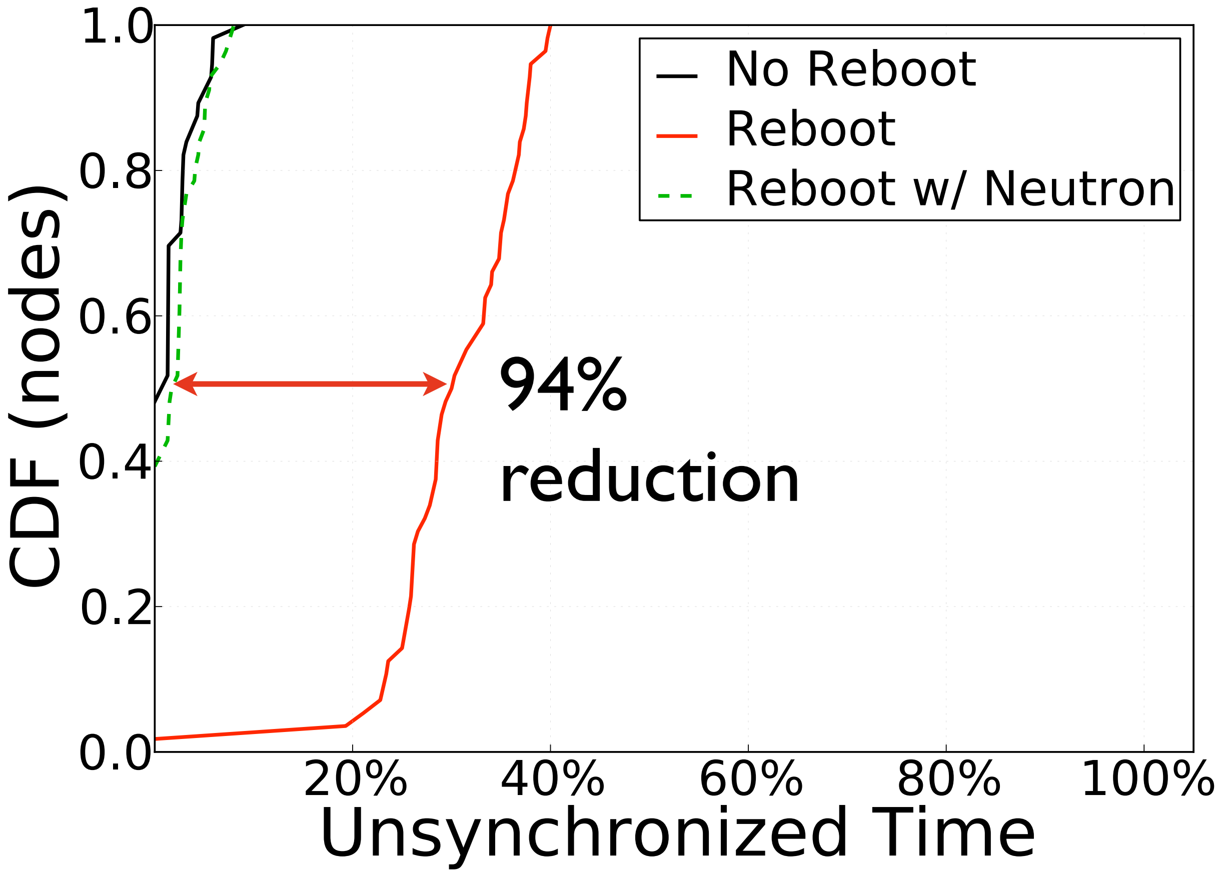
Kernel Reboot: FTSP



Kernel Reboot: FTSP



Kernel Reboot: FTSP



Fault Isolation

- CTP/FTSP persist on an application fault
- Application data persists on a kernel fault

Cost (ROM bytes)

	Safe TinyOS	Neutron	Increase	Increase
Blink	6402	8978	2576	40%
BaseStation	26834	31556	4722	18%
CTPThreadNonRoot	39636	43040	3404	8%
TestCollection	44842	48614	3772	8%
TestFtsp (no threads)	29608	30672	1064	3%

Customized reboot
code is small, still fits on these devices

Cost (reboot, ms)

	Node	Kernel	Application
Blink	12.2	11.4	1.16
BaseStation	22.1	14.1	9.18
CTPThreadNonRoot	15.6	15.5	1.01
TestCollection	15.6	15.5	0.984
TestFtsp (no threads)	14.8	-	-

Cost (reboot, ms)

Kernel fault: CPU busy
for 10-20 ms

	Node	Kernel	Application
Blink	12.2	11.4	1.16
BaseStation	22.1	14.1	9.18
CTPThreadNonRoot	15.6	15.5	1.01
TestCollection	15.6	15.5	0.984
TestFtsp (no threads)	14.8	-	-

Outline

- Recovery units
- Precious state
- Results
- Conclusion

What’s Different Here

- Persistent data in the OS (RioVista, Lowell 1997)
  - Neutron: no backing store, modify in place
- Microreboots (Candea 2004)
  - Kernel and applications, rather than J2E
  - Doesn’t require a transactional database

What’s Different Here

- Rx (Qin 2007) and recovery domains (Lenharth 2009)
  - Almost no CPU cost in execution, microreboots
- Failure oblivious computing (Rinard 2004)
  - Recover from, rather than mask faults

What's Different Here

- Changing the TinyOS toolchain is easy
- Changing the TinyOS programming model isn't (e.g., adding transactions)
  - 90,000 lines of tight embedded code
  - 35,000 downloads/year

Neutron

- Divides a program into recovery units
- Precious state can persist across a reboot
- Near-zero CPU overhead in execution
- Applications survive kernel violations
- Reduces the cost of a violation by 95-99%
- Works on a 16-bit low-power microcontroller

Questions

Diagnosing Faults

At label (2) on August 8, a software command was transmitted to reboot the network, using Deluge [6], in an attempt to correct the time synchronization fault described in Section 7. This caused a software failure affecting all nodes, with only a few reports being received at the base station later on August 8. After repeated attempts to recover the network, we returned to the deployment site on August 11 (label (3)) to manually reprogram each node....

...In this case, the mean node uptime is 69%. However, with the 3-day outage factored out, nodes achieved an average uptime of 96%.

"Fidelity and Yield in a Volcano Monitoring Sensor Network." Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation* (OSDI 2006), Seattle, November 2006.

Given the logistics of our deployment we weren't really able to do much information gathering once Deluge went down in the field, as we simply couldn't communicate with the testbed until the problem was resolved and it was more important to us, at the time, to get our system back on its feet than to debug Deluge. Note that I believe that the reboots were really more the *symptom*, not the *cause* of the Deluge issue (I think)....

....Anyway, in short this is a long way of saying that we actually have no idea what happened to Deluge.

From: challen@eecs.harvard.edu
Subject: **Re: reventador reboots**
Date: July 18, 2009 9:15:26 AM PDT